Cours de Bases de données

INTRODUCTION

I. HISTORIQUE

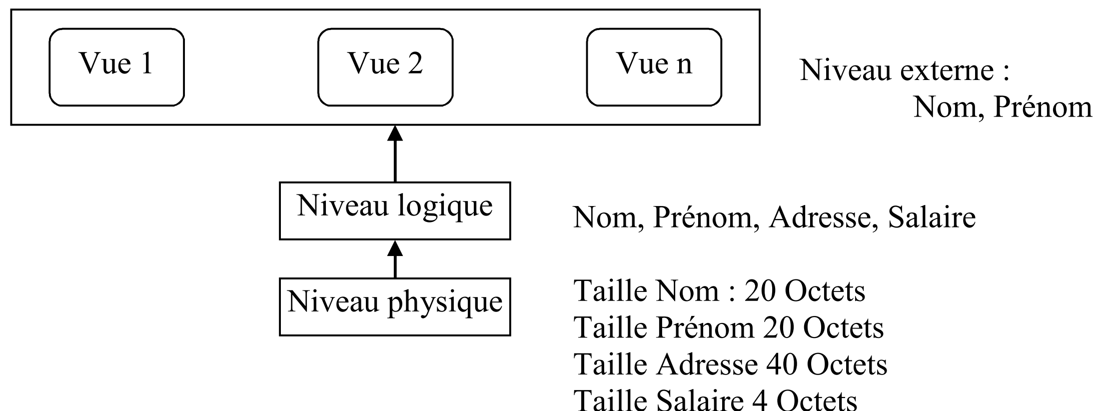
Dès le début de l'informatique, on a voulu construire des systèmes pour effectuer des calculs (équations différentielles, calcul matriciel, ...). Aujourd'hui, la tendance actuelle est la gestion de grandes quantités d'informations. Cela revient à stocker des données et manipuler ces données. Notons que les données peuvent être de natures diverses et les opérations plus ou moins compliquées.
Exemples d'applications :
Applications de gestion (paye, stock, ...), applications transactionnelles (banque, réservation...), applications de documentation (bibliothèque, cartographie, ...), Génie logiciel (ateliers de génie logiciel), Ingénierie (PAO, CAO, ...).

II. Fonctionnalités d'un SGBD

Les problèmes sont de stocker des données (BD) et de manipuler des données (SGBD).

- Gestion du stockage : faire face à des tailles énormes de données.
- Persistance: Les données «survivent» aux programmes qui les créent.
- Fiabilité : Mécanismes de reprise sur pannes (logiciel ou matériel)
- Sécurité - Confidentialité : Droits d'accès aux données
- Cohérence : Contraintes d'intégrité contrôle de concurrence: Conflits d'accès. Répercussions sur la cohérence
- Interfaces homme – machine : Convivialité + différents types d'utilisateurs
- Distribution : Données stockées sur différents sites
- Optimisation : Transferts MC-MS

2.1) Niveau d'abstraction des données



2.2) Instance et schéma

C'est tout comme les notions de « type » et de « variable » dans les langages de programmation standard.
Schéma: C'est la structure logique de la base de données.
Exemples : Ensembles de clients, de produits et de fournisseurs.

Instance : C'est le contenu effectif de la base de données à un instant donné.

III. Principes de base

Indépendance physique: Les applications manipulant la base au niveau logique ne doivent pas être réécrites si la structure physique est modifiée.

Indépendance logique : Une modification au niveau logique n'implique pas forcement une modification des applications utilisant le niveau externe.

3.1) Modèle de données

C'est un ensemble d'outils permettant de définir les schémas et instances, et de définir les opérations possibles sur les instances (et les schémas).

➢ Le *modèle relationnel* permet une description «tabulaire» des données :

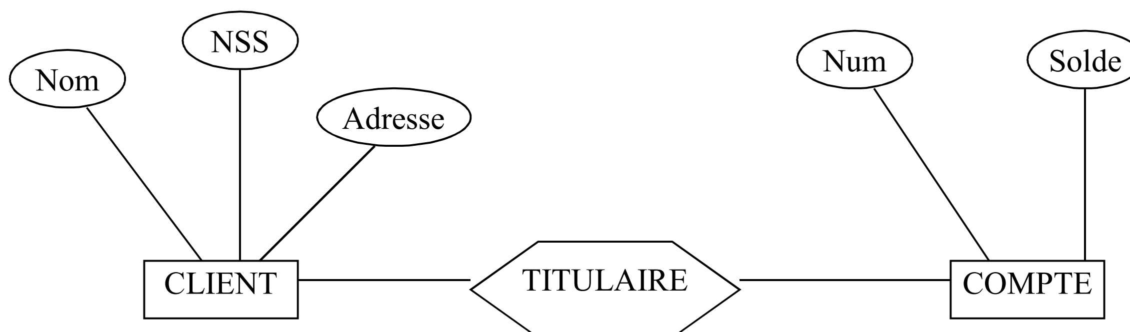
CLIENT

n	resse	n_Cte
n Dupont	ue neuve	5
ert Martin	d de la vieille	6

COMPTE

n_Cte	de
5	00
6	00

➢ Le modèle *Entité-Association* permet une description graphique :



Architecture fonctionnelle d'un SGBD

Niveau physique

Gestion de MS, de concurrence, de la fiabilité, transferts MG-MS, structure d'index, exécution des programmes objet, optimiseur de requêtes

Niveau logique

Sécurité (confidentialité), Intégrité (en partie)

Niveau externe

Environnement de programmation, Interfaces graphiques

IV. Utilisateurs d'un SGBD

Le SGBD offre deux types de langages :

➢ LDD: Définition des données (schéma)

➢ LMD: Manipulation des données (requête et mise à jour)

Administrateur:

Définition du schéma logique, définition des structures de stockage et des méthodes d'accès, autorisations, spécifications des contraintes, maintenance de la performance…

Concepteur et programmeur d'applications

Généralement, il est informaticien, il connaît bien le SGBD et il connaît au moins le LMD et un ou plusieurs langages de programmation.

Utilisateur "naïf"

Secrétaire, caissière, ...

V. Conception d'une BD

On peut la découper en plusieurs étapes :

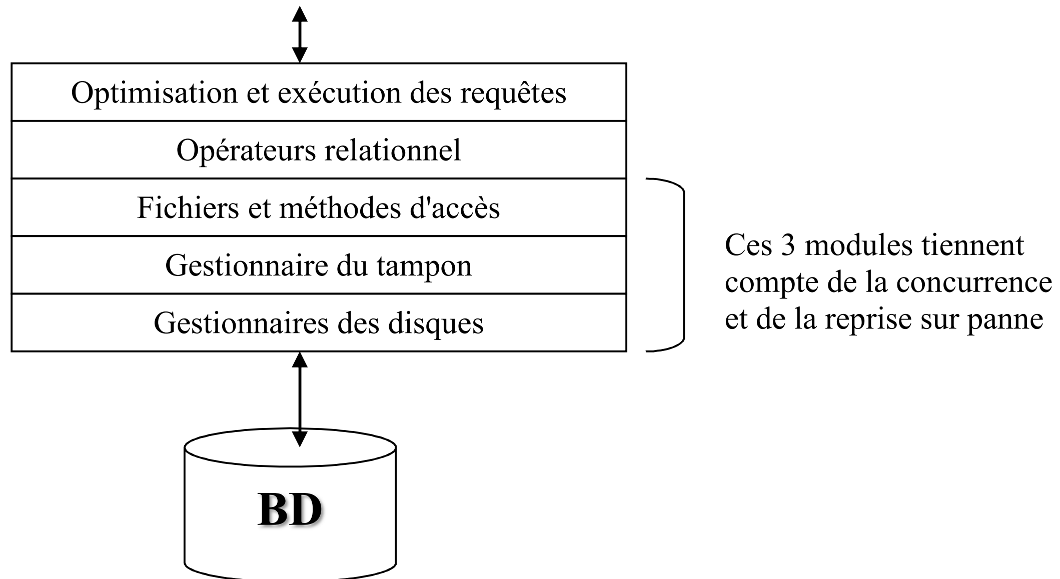
1. Analyse des besoins

2. Description conceptuelle
3. Conception logique (schéma logique)
4. Conception physique

A savoir que les 2 premières phases sont indépendantes du SGBD et que le passage de la phase 2 à la phase 3 peut être en partie automatisé.

6.1) Architecture d'un SGBD

Requêtes



I. Structure d'une B.D. Relationnelle

Les données sont structurées en <u>tables (relations</u>) Etant donnés les ensembles $A_1,, \ldots A_n$,une relation *r* est un sous ensemble de A_1 x A_2 x ... x A_n.. Une relation est un <u>ensemble de n-uplets</u> (ou tuples) de la forme <a1,... an> avec ai $\in$ *Ai*.

<u>Exemple</u> : On a trois ensembles : Nom, Num_Cte et Rue avec
Nom = {Durand, Dupont, Dupond}
Num_Cte = {123, 124, 235 , 226}
Rue = {Neuve, vieille, Courte }
Alors
{< Dupont, 123, Neuve >,
<Dupont, 124, Neuve>,
<Dupond, 235, Neuve >,
<Durand, 123, Vieille> }
est une relation sur Nom x Num_Cte x Rue

1.1) Schéma de relation

Une table est une relation (au sens mathématique) qui a un nom
$A_1,, \ldots ,A_n$ sont des <u>attributs</u>
$R(A_1,\ldots ,A_n)$ est un <u>schéma de relation</u>. (*R* est le nom du schéma de la relation)

On note *Att (R)* pour désigner <u>l'ensemble des attributs</u> de *R*. L'<u>arité</u> de *R* est la cardinalité de *Att(R)*. Le <u>domaine</u> de *Ai* (noté *dom(Ai))* est l'ensemble des valeurs associées à *Ai*. Cet ensemble peut être fini ou non

1.2) Instance de relation

p	n	n_Cte	
	pont		ve
	pond		ve
	and		ille

Att(Emp) = {Nom, Num_Cte Rue}
Arité(Emp) = 3
Dom(Num_Cte) = les entiers naturels (infini)
Dom(Nom) = chaînes de moins de 20 caractères (fini)

1.3) Langages de requête

Ce sont les langages qui permettent « d'interroger » la BD.

Langages relationnels "purs"

- Algèbre relationnelle
- Calcul relationnel par n-uplet
- Calcul relationnel par domaine

Langages pratiques

- SQL (Structured Query Language)
- QUEL (Query Language)
- SEQUEL (Structured English as a Query Language)
- QBE (Query By Example)

II. ALGEBRE RELATIONNELLE

On peut la définir en six opérations de base dont certaines sont unaires, d'autres sont binaires :

1. Projection
2. Sélection
3. Union
4. Différence
5. Produit cartésien
6. Renommage

2.1) Projection

Notation : $\pi\ A_1,\ldots,A_k\ (r)$ où r est le nom de relation et $\forall 1 \leq i \leq k\ \ A_i \in Att(r)$.

Le résultat de cette opération est une relation avec *k* colonnes

Exemple de projection :

On veut extraire les noms des employés de la relation «Emp» ci-dessous :

p	n	n_Cte	
	pont		ve
	pont		ve
	and		ille

π Nom (Emp) = Dupont, Durand

2.2) Sélection

Notation : **σ *Cond (r)*** où r est le nom d'une relation et *Cond* est une condition de la forme

1. $Att_i\ \theta\ Att_j$ ou $Att_i\ \theta$ constante avec $\theta \in \{<, \leq, =, \geq, >, \neq\}$, ou bien
2. une conjonction ($\wedge$) ou une disjonction (V) de conditions

Le résultat est une relation qui contient tous les n-uplets de *r* qui satisfont la condition *Cond*
Exemple de sélection :
On veut avoir les informations concernant les employés dont le nom est Dupont

p	n	n_Cte	:
	pont		ve
	pont		ve
	and		ille

σ *(Nom=Dupont)* =

n	n_Cte	:
pont		ve
pont		ve

2.3) Union, Différence et Intersection

- Opérations ensemblistes classiques
- Notation : $r \cup s$; $r - s$; $r \cap s$
- $r \cup s = \{t \mid t \in r \text{ ou } t \in s\}$
- $r - s = \{t \mid t \in r \text{ et } t \notin s\}$
- $r \cap s = \{t \mid t \in r \text{ et } t \in s\}$
- Opérations binaires
- Il faut que $Att(r) = Att(s)$

			s		

; =

r ∪ s =

r ∩ s =

2.4) Produit cartésien

Notation : **r x s** avec $r \times s = \{ tv \mid t \in r \text{ et } v \in s \}$

Où tv est la concaténation des tuples *t* et *v*.

Cette opération n'est pas définie si $Att(r) \cap Att(s) \neq 0$.
$Att(r \times s) = Att(r) \cup Att(s)$
Exemple de Produit cartésien :

	A	B	s	C	D	E
	α			α	0	-
	ß	2		ß	0	-

s	A	B	C	D	E
	α	1	ι	0	-
	α	1	;	0	
	ß	2	ι	0	-
	ß	2	;	0	

2.5) Renommage

Notation : ρ ***Atti* → *Att'i (r)*** .
Ceci permet de renommer l'attribut *Atti* par *Att'i.*
Le résultat est la relation *r* avec un nouveau schéma

Exemple de Renommage :

	A
	0
	0

→B (r) =

	B
	0
	0

2.6) Composition des opérateurs

On peut appliquer un opérateur de l'algèbre au résultat d'une autre opération.
Exemple : $\pi_A\ (\sigma_{B=20}(r))$. On dit que l'algèbre relationnelle est un langage fermé car chaque opération prend une ou deux relations et retourne une relation.
Soient les schémas de relation *Tit(Id, Nom, Adresse)* et *Cte(Num, Solde, Id_Tit).* Le compte de numéro *Num* appartient au client identifié par *Id_Tit.* On veut avoir (1) le numéro, (2) le solde et (3) le nom du titulaire de chaque compte débiteur.

Tit

Id	Nom	Adresse
A25	Dupont	Rue neuve
B212	Durand	Rue vieille

Cte

Num	Solde	Id_Tit
120	5234.24	A25
135	-100	A25
275	230	B212

1. *Cte* x *Tit* retourne une relation qui associe à chaque tuple de *Cte,* tous les tuples de *Tit.*
2. σ *Id=Id_Tit (Cte* x *Tit)* élimine les tuples où le compte n'est pas associé au bon titulaire.
3 σ *So1de < 0 (*σ *Id=Id_Tit(Cte* x *Tit))* retient les comptes débiteurs
4. $\pi_{Nom,\ Num,\ Solde}$ (σ Solde<0 (σ *Id=Id_Tit (Cte* x *Tit)))* élimine les attributs non demandés

Comment aurait-on pu faire si dans *Cte* on avait *Id* au lieu de *Id_Tit* comme nom d'attribut ?

2.7) Jointure

Notation : ***r ixi* s** *Att(*r ixi s) = *Att(r)* U *Att(s)*
Résultat : Soient tr ∈ *r* et *ts* ∈ s. *trts* ∈ *r* ixi s SSI ∀A ∈ *Att(r)* ∩ *Att(s) tr.A = ts.A*
Exemple de Jointure

	A	B
	α	0
	α	5
	β	|

s

B	C
0	-
1	-

i s =

A	B	C
α	10	+
β	1	-

➢ Noter que le même résultat peut être obtenu comme suit
1. *temp₁* $:=\rho_{B\to B1}(s)$

2. $temp_2 := r \times temp_1$
3. $temp_3 := \sigma_{B=B1}(temp_2)$
4. $res := \pi_{A,B,C}(temp3)$

- La jointure n'est pas une opération de base de l'algèbre relationnelle

III. CALCUL RELATIONNEL par n-uplet

- Les requêtes sont de la forme $\{t \mid P(t)\}$.
- C'est l'ensemble des n-uplets tels que le prêdicat *P(t)* est vrai pour *t*.
- t est une variable n-uplet et *t [A]* désigne la valeur de l'attribut *A* dans *t*
- $t \in r$ signifie que *t* est un n-uplet de *r*
- *P* est une formule de la logique de premier ordre

3.1) Rappel sur le calcul des prédicats

- Des ensembles d'attributs, de constantes, de comparateurs {<, … }
- Les connecteurs logiques 'et' $\wedge$, 'ou' V et la négation $\neg$
- Les quantificateurs $\exists$ et $\forall$
- $\exists$ t $\in$ r (Q(t)) : Il existe un tuple t de r tel que Q est vrai
- $\forall$t $\in$ r(Q(t)) : Q est vrai pour tout t de r

3.2) Exemples de requêtes

Considérons les schémas de relations suivants :
Film (Titre, Réalisateur, Acteur) instance *f*
Programme (Nomciné, Titre, Horaire) instance *p*

f contient des infos sur tous les films et *p* concerne le programme à Bordeaux

- Les films réalisés par Bergman

{t | t $\in$ *f* $\wedge$ *t* [Réalisateur] = "Bergman" }

- Les films où Jugnot et Lhermite jouent ensembles

{ t | t $\in$ *f* $\wedge$ $\exists$ s $\in$ *f* (t[Titre] = s[Titre] $\wedge$ t[Acteur] = "Jugnot" $\wedge$ s[Acteur] = "Lhermite") }

- Les titres des films programmés à Bordeaux

{t | $\exists$ s $\in$ *p* (t[Titre] = s[Titre]) }

- Les films programmés à l'UGC mais pas au Trianon

{*t* | $\exists$ s $\in$ *p* (s[Titre] = t[Titre] $\wedge$ s[NomCiné] = "UGC" $\wedge$
$\neg$ $\exists$ u $\in$ *p* (u[NomCiné] = "Trianon" $\wedge$ u[Titre] = t[Titre])) }

- Les titres de films qui passent à l'UGC ainsi que leurs réalisateurs

{ *t* | $\exists$ s $\in$ *p* ($\exists$ u $\in$ *f* (s[NomCiné] = 'UGC" $\wedge$
s[Titre] = u[Titre] = t[Titre] $\wedge$ t[Réal] = u[Réal])) }

3.3) Expressions "non saines"

Il est possible d'écrire des requêtes en calcul qui retournent une relation infinie.

Exemple : Soit NumCte(Num) avec l'instance *n* et la requête {t | $\neg$t $\in$ n} i.e les numéros de compte non recensés. Si on considère que le *Dom(Num) = N,* alors la réponse à cette requête est infinie.

Une requête est *saine* si quelle que soit l'instance de la base dans laquelle on l'évalue, elle retourne une réponse finie. Dépendance du domaine.

3.4) Calcul relationnel par domaine

Les requêtes sont de la forme : $\{ < x_1,\ldots,x_n > \mid P(x_1,\ldots, x_n) \}$

Les xi représentent des variables de domaine.

$P(x_1 ,\ldots , x_n)$ est une formule similaire à celles qu'on trouve dans la logique des prédicats.

Exemple : Les titres de films programmés à l'UGC de Bordeaux

$\{<t> \mid \exists <nc, t, h> \in p(nc = "UGC") \}$

3.5) Relation entre les 3 langages

- Toute requête exprimée en algèbre peut être exprimée par le calcul.
- Toute requête "saine" du calcul peut être exprimée par une requête de l'algèbre.
- Les 3 langages sont donc équivalents d'un point de vue puissance d'expression.
- L'algèbre est un langage *procédurale* (quoi et comment) alors que le calcul ne l'est pas (seulement quoi).

IV. LE LANGAGE SQL

C'est un langage fourni avec tout SGBD relationnel commercialisé. C'est un standard reconnu par l'IS0 depuis 87 (standard donc portabilité). On en est à la version 2 (SQL92) et la version 3 est annoncée pour bientôt. SQL est un LDD et un LMD. Il est aussi utilisé pour définir des vues, les droits d'accès, manipulation de schéma physique…

4.1) Structure de base

Une requête SQL typique est de la forme
SELECT $A_1,, \ldots , A_n$
FROM $r_1,\ldots ,r_m$
WHERE *P*

Les Ai sont des attributs, les rj sont des noms de relations et P est un prédicat.
Cette requête est équivalente à $\pi A_1 ,\ldots, A_n (\sigma_P(r_1 \times \ldots \times r_m))$
La clause SELECT
La clause SELECT correspond à la projection de l'algèbre.

Les titres des films :
SELECT Titre
FROM film
L'utilisation de l'astérisque permet de sélectionner tous les attributs :
SELECT *
FROM film
SQL autorise par défaut les doublons. Pour le forcer à les éliminer, on utilise la clause **DISTINCT :**
SELECT DISTINCT Titre
FROM film
SELECT peut contenir des expressions arithmétiques ainsi que le renommage d'attributs :
SELECT Prix_HT * 1.206 **AS** Prix TTC
FROM produit

La clause WHERE
Elle correspond au prédicat de sélection dans l'algèbre. La condition porte sur des attributs des relations qui apparaissent dans la clause **FROM**

SELECT DISTINCT Titre
FROM film
WHERE Réalisateur "Bergman" **AND** Acteur = "Stewart"

SQL utilise les connecteurs **AND, OR** et **NOT**. Pour simplifier la clause WHERE, on peut utiliser la clause **BETWEEN.**

SELECT Num
FROM compte
WHERE Solde **BETWEEN** 0 **AND** 10000

La clause FROM

Elle correspond au produit cartésien de l'algèbre.

Le titre et le réalisateur des films programmés à l'UGC de Bordeaux.

SELECT Titre, Réalisateur
FROM film, programme
WHERE film.titre = programme.titre **AND** programme.NomCiné = "UGC"

Les variables n-uplets

Elles sont définies dans la clause **FROM**

SELECT Titre, Réalisateur
FROM film AS f, programme AS p
WHERE f.titre = p.titre **AND** p.NomCiné = "UGC"

Soit Emp (Id, Nom, Id_chef)

SELECT el.Nom, e2.Nom **AS** Nom_Chef
FROM emp el, emp e2
WHERE el.Id_chef = e2.Id

La clause ORDER BY

SQL permet de trier les résultats de requête

SELECT *
FROM programme
WHERE NomCiné="UGC"
ORDER BY Horaire **ASC**, Titre **DESC**

4.2) Opérateurs ensemblistes

SELECT ...
…
UNION/ INTERSECT/ EXCEPT
SELECT ...

Attention : Ces opérations éliminent les doublons, pour pouvoir les garder, utiliser à la place **INTERSECT ALL**... Si t apparaît m fois dans r et n fois dans s alors il apparaît :

- m + n fois dans r **UNION ALL** s
- min(*m, n)* fois dans *r* **INTERSECT ALL** s
- max(0, *m* - *n)* fois dans *r* **EXCEPT ALL** s

4.3) Les fonctions d'agrégats

Ce sont des fonctions qui agissent sur des ensembles (multi-ensembles) de valeurs :

AVG : la valeur moyenne de l'ensemble

MIN : la valeur minimale
MAX : la valeur maximale
SUM : le total des valeurs de l'ensemble
COUNT : le nombre de valeur dans l'ensemble

SELECT COUNT(Titre) **FROM** Programme
Cette requête retourne le nombre de films programmés à Bordeaux.
Attention : Un même titre peut être compté plusieurs fois s'il est programmé à des heures différentes et dans des salles différentes.

SELECT COUNT(**DISTINCT** Titre) **FROM** Programme

Agrégats et GROUP BY

Le nombre de films programmés dans chaque salle :

SELECT NomCiné, **COUNT** (**DISTINCT** Titre)
FROM Programme
GROUP BY NomCiné

Les attributs qui apparaissant dans la clause **SELECT** en dehors des agrégats *doivent* être associés à la clause **GROUP BY**

Agrégats et la clause HAVING

Les salles où sont programmés plus de 3 films :

SELECT NomCiné, COUNT(DISTINCT Titre)
FROM Programme
GROUP BY NomCiné
HAVING COUNT (**DISTINCT** Titre) > 3

Le prédicat associé à la clause **HAVING** est testé après la formation des groupes définis dans la clause **GROUP BY.**

4.4) Requêtes imbriquées

SQL fournit un mécanisme qui permet d'imbriquer les requêtes. Une sous requête est une requête SQL (SELECT-FROM-WHERE) qui est incluse dans une autre requête. Elle apparaît au niveau de la clause **WHERE** de la première requête.

Les films programmés à l'UGC non programmés au Trianon
SELECT Titre
FROM Programme
WHERE NomCiné="UGC" and Titre **NOT IN** (
SELECT Titre
FROM Programme
WHERE NomCiné ="Trianon")

Trouver les comptes dont les soldes sont supérieurs aux soldes des comptes de Durand :
Compte (Num, Solde, NomTit)
SELECT *
FROM Compte
WHERE Solde > **ALL** (
SELECT Solde
FROM Compte
WHERE NomTit = "Durand")

En remplaçant **ALL** par **SOME**, on obtient les comptes dont les soldes sont supérieur au solde d'au moins un compte de Durand.

Les cinémas qui passent tous les films programmés àl'UGC

```
SELECT              NomCiné
FROM                programme pi
WHERE NOT EXISTS (
(SELECT DISTINCT Titre
FROM                      programme
WHERE                     NomCiné-"UGC")
EXCEPT
(SELECT DISTINCT Titre
FROM programme p2
WHERE pl.NomCiné = p2.NomCiné))
```

Test d'absence de doublons

La clause **UNIQUE** permet de tester Si une sous requête contient des doublons.

Les titres de films programmés dans une seule salle et un seul horaire :

```
SELECT              p.Titre
FROM                programme p
WHERE UNIQUE (
SELECT                    pl.Titre
FROM                      programme p1
WHERE                     p.Titre = pl Titre)
```

Les relations dérivées

Titulaire (Nom, Adresse)
Compte (Num, Solde, NomTit)

Donner le solde moyen des comptes de chaque personne ayant un solde moyen > à 1000

```
SELECT              NomTit, SoldeMoyen
FROM (
SELECT                    NomTit, AVG(Solde)
FROM                      Compte
GROUP BY NomTit )
AS Result (NomTit, SoldeMoyen)
WHERE SoldeMoyen > 1000
```

Noter qu'on aurait pu exprimer cette requête en utilisant la clause **HAVING**

4.5) Les vues

Equivalent à une requête Access. Une vue peut être considérée comme une relation quelconque lors de l’expression des requêtes. Une vue est une relation virtuelle dans le sens où elle ne contient pas effectivement des « tuples ». Elles permettent de définir des relations *virtuelles* dans le but de :

- Cacher certaines informations à des utilisateurs,
- Faciliter l'expression de certaines requêtes,
- Améliorer la présentation de certaines données.

Une vue est définie par une expression de la forme :

CREATE VIEW V **AS** requête

Requête est une expression quelconque de requête et V est le nom de la vue.

Emp (NumE, Salaire, Dept, Adresse)
CREATE VIEW EmpGen **AS** (
SELECT NumE, Dept, Adresse
FROM Emp)
Toutes les informations concernant les employés du département 5 :
SELECT *
FROM EmpGen
WHERE Dept = 5

4.6) Modification des relations

Suppression :
Supprimer tous les employés du département 5 : **DELETE FROM** Emp
WHERE Dept = 5

Supprimer du programme tous les films programmés à l'UGC où un des acteurs est DiCaprio :
DELETE FROM programme
WHERE NomCiné = "UGC' **AND EXISTS** (
SELECT Titre
FROM film
WHERE programme.Tite = film.Titre **AND** film.Acteur = "DiCaprio")

Supprimer les comptes dont le solde est < à la moyenne des soldes de tous les comptes :
DELETE FROM compte
WHERE Solde < (**SELECT AVG** (Solde) **FROM** compte)
Problème : Si les n-uplets sont supprimés un à un de la relation compte, alors à chaque suppression, on a une nouvelle valeur de **AVG** (Solde). La solution de SQL est de d'abord, calculer **AVG**(Solde) et ensuite de supprimer les tuples satisfaisant le test sans recalculer à chaque fois la nouvelle valeur de **AVG** (Solde).
Insertion
Insérer un n-uplet dans la relation "compte" :
INSERT INTO compte (Num, Solde, NomTit) **VALUES** (511,1000, "Dupont")
ou bien **INSERT INTO** compte **VALUES** (511,1000, «Dupont")

Insère un n-uplet avec un solde *inconnu.*
INSERT INTO compte **VALUES** (511, NULL, "Dupont")
ou bien **INSERT INTO** compte(Num, NomTit) **VALUES** (511, "Dupont")

Les 2 dernières MAJ sont équivalentes sauf si une valeur par défaut du Solde a été spécifiée lors de la définition de la table compte.

Supposons qu'on a crée une relation TitMoy (NomTit, Moyenne) qui doit contenir le nom des clients de la banque ainsi que la moyenne des soldes de leurs comptes.
INSERT INTO TitMoy (NomTit, Moyenne)
SELECT NomTit, **AVG**(Solde)
FROM compte
GROUP BY NomTit
Update
Rajouter 1% à tous les comptes dont le solde est inférieur à 500 :
UPDATE compte
SET Solde = Solde * 1.01
WHERE Solde ≤ 500
La condition qui suit la clause **WHERE** peut être une requête SQL.

SQL EN TANT QUE LDD

- Le schéma des relations
- Les domaines des attributs

- Les contraintes d'intégrité
- La gestion des autorisations
- La gestion du stockage physique
- Les index associés à chaque relation

5.1) Domaines

- **char(n)** : chaîne de caractères de taille fixe n
- **varchar(n)** : chaîne de caractères de taille variable mais inférieure à n
- **int** : Entier (un sous ensemble fini des entiers, dépend de la machine)
- **smallint** : Entier. Sous ensemble de int
- **numeric(p,d)** : Réel codé sur *p* digits et max *d* digits pour partie à droite de la décimale.
- **real** : Un réel flottant.
- **date** : YYYY-MM-DD (année, mois, jours)
- **time** : HH :MM :SS (heure, minute, seconde)

Les valeurs nulles (NULL) sont possibles dans tous les domaines. Pour déclarer qu'un attribut ne doit pas être nul, il faut rajouter **NOT NULL**

- **CREATE DOMAIN** nom-client char(20)

5.2) Création des tables

- On utilise la clause **CREATE TABLE**

CREATE TABLE compte (
Num **int NOT NULL**,
Solde **int**,
NomTit **varchar(20)**)

- Rajout de contraintes

CREATE TABLE compte (
Num **int NOT NULL**,
Solde **int DEFAULT** 0,
NomTit **varchar(20),**
PRIMARY KEY (Num),
CHECK (Num > 1))

- En SQL92, si un attribut est clé alors il est différent de NULL.

Manipulation de schéma

La commande **DROP TABLE** permet de supprimer une table.
Ex: **DROP TABLE** compte.

Si une vue est définie sur la table compte alors il faut utiliser
DROP TABLE compte **CASCADE**

La commande **ALTER TABLE** permet de modifier le schéma d'une relation.
Exemple : **ALTER TABLE** compte **ADD** Date_ouverture
date
ALTER TABLE compte **DROP** Solde **CASCADE**

Clé étrangère

Soient Personne (NSS, Nom) et Voiture (Matricule, modèle, Proprio).
« Proprio » correspond au NSS du propriétaire. C'est une *clé étrangère* dans le schéma **Voiture** car c'est une clé dans un autre schéma.

CREATE TABLE Voiture (
Matricule **CHAR**(8),
Modele **CHAR**(10),
Proprio **CHAR**(3),
PRIMARY KEY(Matricule),
FOREIGN KEY(Proprio) **REFERENCES** Personne
ON [DELETE | UPDATE] CASCADE |
RESTRICT |
SET NULL
)

CASCADE: Si une personne est supprimée, alors les voitures qu'elle possède sont supprimées.
RESTRICT : Le système refuse la suppression d'une personne s'il y a des voitures qui lui sont rattachées. C'est l'option par défaut.
SET NULL: Si une personne est supprimée, alors l'attribut Proprio prend la valeur NULL.

L'insertion d'une voiture ne peut se faire que si le «proprio » existe dans **Personne** (ou bien valeur nulle).

Valeurs nulles

ployé	Nom	Salaire
	Dupont	10 000
	Martin	NULL

SELECT *
FROM Employé
WHERE Salaire > 12000 *Ne retourne aucun tuple.*
Pareil si la condition est **WHERE** Salaire < 8000

SELECT SUM (Salaire) **FROM** Employé**;** *Retourne 10000*

SELECT COUNT (Salaire**) FROM** Employé**;** *Retourne 2*

SELECT AVG (Salaire**) FROM** Employé**;** *Retourne 10000*
Très différent de **SELECT SUM** (Salaire**) / COUNT** (Salaire**) FROM** Employé *car COUNT prend en compte la valeur NULL donc cela fera 10000/2=5000.*
*En fait c'est équivalent à :***SELECT SUM** (Salaire**) / COUNT** (Salaire**) FROM** Employé
WHERE Salaire **IS NOT NULL**

SELECT COUNT(*) FROM Employé
WHERE Salaire IS NOT NULL; *Retourne 1*

Mise à jour des vues

Personne (Nom, Salaire). Supposons que la table Personne est vide.
CREATE VIEW Gros_Salaire **AS**
SELECT *
FROM Personne
WHERE Salaire > 10000

INSERT INTO Gros_Salaire **VALUES**("Martin", 5000)
L'effet de cette commande est d'insérer dans la table *Personne* le tuple < "Martin", 5000>.
Noter que si l'on fait :
SELECT * FROM Gros Salaire; on n'obtient aucun tuple.

Si à la création de la vue on rajoute l'option :
WITH CHECK OPTION alors l'insertion est refusée.

Les mises à jours des vues sont traduites en des mises à jours des tables sous-jacente. *La traduction n'est pas toujours unique.* Quand nous avons plusieurs manières de traduire une mise à jour alors celle-ci est rejetée. ⇒ Certaines vues ne permettent pas des mises à jour. Il faut une relation biunivoque entre la mise à jour de la vue et la mise à jour de la table.

Jointure externe
Si on fait : Personne IXI Voiture, on n'aura que les personnes qui ont une(des) voiture(s) qui sont dans le résultat.
SELECT *
FROM Personne P **Left Outer Join** Voiture V
ON P.NSS = V.Proprio

Cette requête retourne aussi les personnes n'ayant pas de voiture. Ces tuples auront des *valeurs nulles* pour les champs provenant de Voiture. Si on met juste **Outer Join** alors on aura les personnes sans voitures et les voitures sans Propriétaire.

La jointure est exprimée par : T1 **Inner Join** T2 **On** Condition

Dans l'exemple, si l'on veut joindre Personne et Voiture alors
SELECT *
FROM Personne P **Inner Join** Voiture V
ON P.NSS = V.Proprio
Si l'on mets **Right** à la place de **Left** dans la jointure, alors on aura les voitures sans les personnes. Si on ne mets ni Left ni Right, on aura les voitures et les personnes qui ne sont pas dans la jointure.

5.3) Mécanisme des droits

Soit la table pers (Num, Nom, Adr, Num_Serv, Salaire)

1. Dupont ne peut pas accéder à Pers.
2. Dupont peut lire une partie de Pers mais ne peut rien modifier.
3. Dupont Peut lire un seul tuple (celui le concernant) sans pouvoir le modifier.
4. Dupont peut en plus modifier l'attribut Adr.
5. Dupont peut accéder à l'attribut salaire mais seulement entre 9h et 17h à partir du terminal 25.
6. Dupont peut modifier salaire si celui-ci est inférieur à 8000.
7. Dupont peut modifier la relation s'il est responsable du Num_Serv du tuple qu'il veut modifier.

Les droits dépendent du contenant, du contexte et/ou du contenu.

Les droits dans SQL
SELECT: privilège qu'il faut posséder pour lire une table
INSERT, DELETE, UPDATE: privilèges nécessaires pour mettre à jour une table.
INSERT(X), UPDATE(X): privilège nécessaire pour insérer, mettre à jour l'attribut X.

Octroi et retrait de privilèges

GRANT privilège **ON** objet **TO** utilisateur [**WITH GRANT OPTION**]

REVOKE [GRANT OPTION FOR] privilège **ON** objet **FROM** utilisateur
RESTRICT | CASCADE

Exemples :
GRANT ALL ON TABLE résultat **TO** directeur **WITH GRANT OPTION;**

GRANT INSERT ON TABLE résultat **TO** sec_1**;**

GRANT SELECT, UPDATE(points) **ON TABLE** resultat **TO** prof_1;

Remarque : Un utilisateur peut recevoir le même privilège de plusieurs sources. Cela est utile quand l'une d'elles veut le lui retirer.

Exemple : soit la séquence
A: GRANT SELECT ON TABLE T TOBWITH GRANT OPTION
B: GRANT SELECT ON TABLE T TOCWITH GRANT OPTION
C: GRANT SELECT ON TABLE T TODWITH GRANT OPTION
A: REVOKE SELECT ON TABLE T FROM B
Ni B ni C ne pourront lire T

Utilisation des vues

CREATE VIEW Informations_Perso
AS SELECT *
FROM Employé
WHERE nom = **USER**;

GRANT SELECT, UPDATE(adresse)
ON Informations_Perso
TO PUBLIC

V. EXERCICES : REQUETES SQL

6.1) Exercice 1: Base Cinéma

Soit la base Cinéma ayant le schéma Cinéma (Film, Lieu, Pgme_hebdo). Les schémas de relation sont les suivants

Films(Titre, Réalisateur, Acteur), Lieu(Salle, Adresse, Tel) et Pgme_hebdo(Salle, Titre, jour, horaire). Exprimer les requêtes suivantes en SQL

1. Qui est le réalisateur de "The big Lebowsky" ?
2. Dans quelles salles passe le film "Cris et chuchotements" (On veut le nom de la salle ainsi que son adresse) ?
3. Quels sont les réalisateurs qui jouent dans leurs propres films
4. Donner les paires de personnes (Pi, *P2)* telles que Pi a dirigé *P2* dans un film et *P2* a dirigé Pi dans un autre.
5. Quelles sont les salles où je pourrais voir un film de où joue "J.P Bacri'"?
6. Quelles sont les salles qui programment tous les films de où joue "J.P Bacri"?
7. On veut pour chaque réalisateur, le nombre de ses films qui sont programmés, à condition que ce nombre soit supérieur à 3.
8. Quels sont les réalisateurs qui ont dirigé le plus d'acteurs.
9. Quels sont les réalisateurs qui ont dirigé plus d'acteurs que Bergman et Hitchcok

6.2) Exercice 2: Base Enseignement

Soit la base Enseignement(Etudiant, Cours, Inscrit, Enseignant) où les schémas des relations sont respectivement
Etudiant(<u>NumE</u>, NomE, AgeE)
Cours(<u>NomC</u>, HoraireC, SalleC, EnsC)
Inscrit(<u>NumE, NomC</u>)
Enseignant(<u>NumEn</u>, NomEn, NumDept)

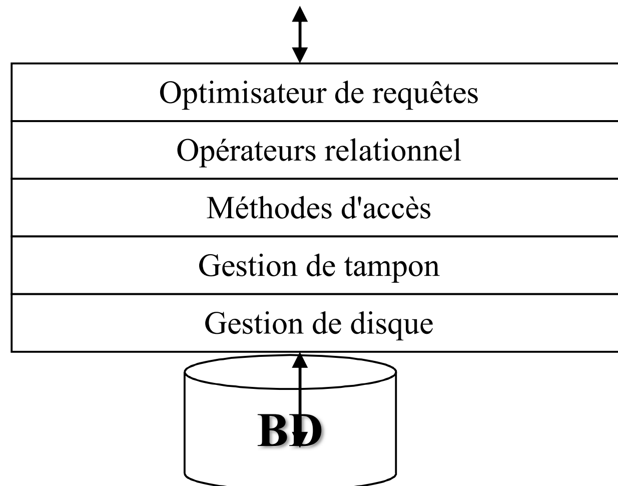
Exprimer les requêtes suivantes en SQL

1. Quels sont les cours ayant lieu en salle 123?
2. Quels sont les cours pour lesquels il y a plus de 5 inscrits ?
3. Quels sont les noms des étudiants inscrits à au moins deux cours prévus aux mêmes horaires ?
4. Quels sont les enseignants qui ont moins de 20 étudiants (en considérant tous les cours qu'ils assurent) ?
5. Quels sont les noms des étudiants ayant le plus d'inscriptions ?
6. Quels sont les étudiants qui ne sont inscrits à aucun cours ?

Fichiers et disques

I. Structure fonctionnelle d'un SGBD

Requêtes



Fichiers et Disques

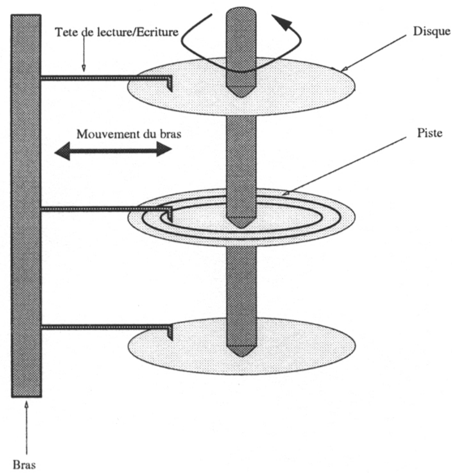
- Lecture : Transfert des données du disque vers la mémoire centrale.
- Ecriture : Transfert de la RAM vers le disque.

Ces 2 opérations sont relativement coûteuses par rapport a d'autres opérations qui sont effectuées en RAM. Mais alors, pourquoi ne pas tout garder en RAM ? Il faut savoir qu'avec 1200 Frs, on a 128Mb de RAM ou 7.5Go de disque. Il faut aussi prendre en compte la volatilité de la RAM, la hiérarchie de stockage RAM pour les données en cours d'utilisation.

On en déduit qu'il vaut donc mieux utiliser des disques durs pour stocker toute la BD, et des bandes magnétiques ou des CD ROM pour les archives.

LES DISQUES

Le principal avantage des disques est l'accès direct aux données contre un accès séquentiel pour les bandes. Les données sont stockées et donc accédées par blocs ou *pages* entières. Contrairement aux RAM, le temps d'accès aux données sur les disques dépend de l'emplacement. Il faut donc bien placer les données pour optimiser les transferts.



Les disques ont une vitesse de rotation (ex: 90 tr/s).

Les pistes de même diamètre forment un cylindre.

Une seule tête peut lire ou écrire.

La taille d'un bloc est un multiple de la taille d'un secteur.

2.1) Paramètres d'un disque

Temps d'accès a un bloc (TAB):

- *Temps de positionnement* (TP)*:* temps nécessaire pour positionner la tête de L/E sur la bonne piste. Il varie entre 1 et 10 ms.

- *Latence de rotation (LR):* temps nécessaire pour positionner la tête de L/E au bon endroit sur la piste. C'est ½ tour de disque en moyenne. Il dépend donc de la vitesse de rotation du disque (exemple pour 90 tr/s : LR= 1/90 /2= 0.0056). Varie entre 0 et 10 ms.

- *Temps de transfert (TT):* temps du transfert du disque à la RAM (ou inversement). Le débit du transfert est de l'ordre de 5 Mo/ms.

Pour optimiser les transferts, il faut optimiser TP et LR en placant correctement les données sur le disque.

Exemple de caractéristiques de disques durs :

Seagate Hawk 2XL : 2,l5Go. TR = 5.55 ms. TP = 9 ms. Passer d'une piste a une piste voisine = l ms. Le temps de positionnement maximum = 22 ms. 512 octets par secteur. 4559 cylindres. 4 disques double face. Le débit du transfert TT = 5 Mo/s.

High performance Barracuda : 9 Go. TR = 4.l ms. TP = 8.5 ms. TT = lO Mo/s.

2.2) Placement des pages sur disque

Le concept du bloc "suivant" :

1. Mettre les blocs (successifs) sur la même piste,
2. blocs sur le même cylindre, ensuite
3. blocs sur le cylindre adjacent

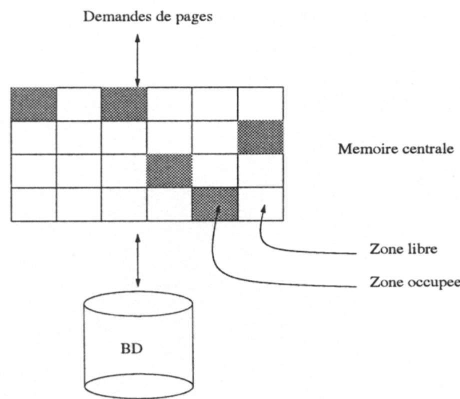
Les blocs d'un fichier doivent être stockés *séquentiellement* sur le disque ("suivant") pour minimiser **TP** et **LR**. Lorsque l'on veut faire un balayage séquentiel d'un fichier, le fait de pouvoir préparer (pré-fetch) plusieurs pages a la fois représente un gain considérable. L'arrangement des fichiers sur les disques est une tache supportée par une couche basse du SGBD (en collaboration avec le système d'exploitation). Elle communique avec la couche juste au dessus par des appels de pages.

1	6	7	.
2	5	.	.
3	4	.	.

Zone Id (identificateur de la zone du tampon)
< 3, 1 > : Page Id (adresse de la page sur le disque)
Page ou bloc : généralement, cela correspond à un secteur (ou alors à un multiple de secteurs). Au niveau du disque : **Page Id < n° plateau, face, n° piste, n° secteur >**

II. GESTIONNAIRE DE TAMPON

3.1) Principes



On a une table de < *zoneId, pageId*>

Si une page n'est pas disponible, le gestionnaire de tampon (GT) émet une demande au gestionnaire de disque. Au retour, s'il n'y a plus de zone libre, le GT *choisit* une zone occupée par une page pour y mettre la nouvelle page. Le critère du choix de la page à remplacer a un effet sur l'optimisation des transferts.

Si les demandes de pages peuvent être prédites, alors on peut charger plusieurs d'une passe (ex: scan séquentiel). Si la page à remplacer a été modifiée, alors il faudra réécrire sa nouvelle version sur le disque. On utilise un bit pour marquer les zones modifiées.

3.2) Critère de remplacement des pages :

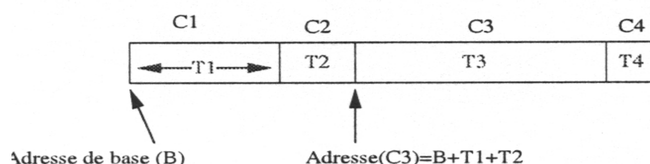
Si le tampon est rempli et on veut charger une nouvelle page alors il faut en choisir une qu'il faudra remplacer par la nouvelle (de préférence, une qui n'a pas été modifiée car on n'a pas à la réécrire sur le disque). Après pour choisir entre celles qui n'ont pas été modifiées, on utilise plusieurs techniques: LRU (Least Recently Used), MRU (Most Recently Used) FIFO (First In First Out), LIFO (Last In First Out).

III. FORMATS D'ENREGISTREMENTS

Les enregistrements sont de taille fixe ou de taille variable. Notons au préalable que le SGBD gère un catalogue qui permet de stocker des informations sur les fichiers (ex: le nom des champs, leurs tailles, …).

4.1) Taille fixe

Dans ce cas, étant donnée l'adresse d'un enregistrement, on peut accéder directement a un champ particulier.

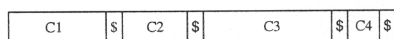


dresse d'un enregistrement est spécifiée le num plateau, num piste, num bloc, ı Secteur

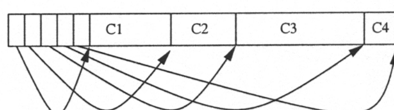
Exemple : 20 caractères pour le nom... Il suffit alors que d'avoir juste le début de l'enregistrement (tuple = enregistrement). Personne, F1 Taille tuple = 40 octets (taille nom : 20 et taille prénom : 20).
Adresse d'un enregistrement < adresse de page , offset > (connaissance de la position de chaque champs) :

4.2) Taille variable

Deux alternatives :



On lit jusqu'au séparateur $ puis ore au $ puison arrivera à C3.



On utilise des pointeurs vers le ut et la fin des champs.

La 2ème alternative est plus intéressante surtout pour la gestion des valeurs **Nulles** : Si un champ est NULL alors les pointeurs de début et de fin associées à ce champ sont égaux (2 champs = 3 pointeurs).

IV. FORMAT DES PAGES

5.1) Enregistrements de taille fixe

- *Organisation contiguë* : Le problème est d'en cas de suppression, il faut réorganiser toute la page. Mais il suffit de consulter N (nbre de pages) pour savoir s'il y a de l'espace libre.

- *Organisation dynamique* : en cas de suppression, on a de la place libre mais il faut écrire les en-têtes (compteur M sur le nombre d'enregistrement). Et pour savoir s'il y a de l'espace libre, on doit donc lire le tableau.

5.2) Enregistrements de taille variable

On ne connaît pas la taille de la liste des pointeurs…

V. FICHIERS NON TRIES (Tas)

C'est la structure de fichier la plus simple. La taille varie au gré des insertions et suppressions des enregistrements. Les pages (blocs) sont allouées / désallouées en conséquence. Pour pouvoir gérer les opérations sur les enregistrements, nous devons garder :

- les références des pages du fichier,
- l'espace libre dans chaque page,
- l'emplacement des enregistrements dans chaque page.

L'entrée de chaque page peut aussi contenir le nombre d'octets libres dans chaque page.

VI. LES INDEX

Les fichiers tas permettent de retrouver un enregistrement en :
- spécifiant le *IdE,* *(avec IdE : identificateur d'enregistrement)*
- ou en balayant tout le fichier.

Généralement en BD, on recherche par valeur. Souvent, on veut les enregistrements selon la valeur de certains champs. Pour ce type de recherche, les fichiers tas ne sont pas adéquats. Les index sont des fichiers "annexes" permettant de répondre efficacement aux recherches basées sur des *valeurs.*

LE CATALOGUE DU SYSTEME

Il fait partie de la base et il est géré comme le reste. Pour avoir la liste des champs de la base, il suffit de faire : **SELECT** *
FROM cat_att *(avec cat_att une table du catalogue)*

Pour chaque index, il contient sa structure (ex: arbre B+) ainsi que sa clé.
Pour chaque relation, il contient :
(1) son nom,
(2) le fichier associé,
(3) la structure de fichier (indexé ou tas),
(4) le type et le nom des attributs,
(5) le nom de(s) l'index,
(6) les contraintes d'intégrité.
Pour chaque vue, (1) son nom et (2) sa définition.

Des statistiques, les droits d'accès, la taille de la zone tampon,

Att_Cat(att_name, rel_name, type, position)

Att_name	Rel_name	type	position

att_name	att_name	char (20)	1
Rel_name	att_name	char (20)	2
type	att_name	char (20)	3
position	att_name	INTEGER	4
NumE	Etudiants	INTEGER	1
Nom	Etudiants	varchar (20)	2
…	dots	…	…

VII. CONCLUSION

Le gestionnaire de tampon *charge* les pages dans la RAM. Les pages restent dans le tampon jusqu'à ce qu'il n'y ait aucun processus qui ne les utilise. Elles sont *réécrites sur* le disque après qu'elles soient libérées et quand la zone qu'elles occupent est choisie pour un remplacement. Le choix de la zone à libérer est dictée par différentes techniques possibles (LRU, ...). Les enregistrements a taille variable permettent de bien gérer les valeurs nulles.

Le gestionnaire de fichier est chargé de la gestion des pages pour chaque fichier. Les index permettent de faire des recherches basés sur des valeurs. Le catalogue contient la description de la structure physique des relations, attributs, fichiers, index, ...

Les transferts se font donc par page (ou par bloc).

I. ORGANISATION DES FICHIERS

E

Une *organisation de fichier est* une manière de disposer les enregistrements dans un fichier stocké sur le disque. Un fichier peut être accédé et modifié de différentes façons, et une organisation de fichier peut être *bonne* pour un type d'accès et *mauvaise* pour un autre type.
Un fichier trié sur le nom des employés n'est pas une bonne organisation quand on veut avoir les employés ayant un salaire supérieur à 100. Un SGBD offre plusieurs organisations possibles. C'est a l'administrateur que revient le choix de l'organisation adéquate. Les fichiers triés sont meilleurs pour les requêtes mais la mise à jour est plus compliquée à gérer.

1.1) Modèle de coût

Pour pouvoir comparer différentes organisations, il nous faut un modèle de coût. Pour cela, on va considérer les paramètres :

- P: Le nombre de pages contenant des données,
- E: Le nombre d'enregistrements par page,
- T: Le temps "moyen" pour lire ou écrire une page,

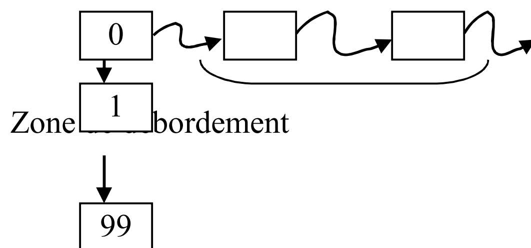
Ainsi le coût d'une opération est exprimée en fonction de ses paramètres.
Dans ce modèle *simplifié,* on ne considère pas le coût **C** des traitements en unité centrale (typiquement C = T/25)

1.2) Opérations et Organisations considérées

- Balayage : (Scan) parcourir tout le fichier.
- Recherche avec égalité : On cherche les enregistrements ayant un champs X égal a une valeur particulière.
- recherche avec intervalle **:** On cherche les enregistrements ayant un champs X compris dans un intervalle particulier.
- Insertion **: (1) On** doit identifier la page qui doit contenir l'enregistrement a insérer (2) la modifier en zone tampon (mémoire) et (3) la réécrire sur disque.
- Suppression **:** Suppression d'un enregistrement identifié par son *IdE.* (1) ramener la page correspondant en mémoire, (2) la modifier ensuite la réécrire sur disque.

Fichiers hachés
Un fichier haché est caractérisé par :
1. un champ particulier appelé *clé de recherche (Rien à voir avec la notion de "clé" dans une table).*
2. une fonction de hachage *h* qui associe a chaque valeur de la *clé de recherche* un entier.
Exemple: *h: String N. h(X) = ASCII(X) modulo(100).*

L'entier associée a une valeur correspond au numéro de la page où l'enregistrement correspondant doit se trouver. Plusieurs valeurs de la clé peuvent être associées au même entier. Nous avons donc une zone de débordement.
Exemple : *h(abc) = h(bca).*



Fichier tas
➢ Balayage : coûte **B x T**. Il faut transférer toutes les pages en mémoire.

➢ Recherche avec égalité :

1. Si on sait qu'il y a un seul enregistrement, alors en moyenne, on lit la moitié du fichier le coût est **P x T / 2**.
2. Si l'on ne sait pas le nombre d'enregistrements, alors le coût est **P x T**. C'est le pire des cas car on balaye tout le fichier ! Dans le meilleurs des cas, la première page lue contient l'information recherchée donc le coût est : **T**.

➢ Recherche avec intervalle **:** On doit balayer tout le fichier : le coût est **B x T**.

➢ Insertion : On suppose que l'insertion se fait a la fin du fichier.
1. On lit la dernière page (1er transfert), puis
2. On la modifie en mémoire et on la réécrit sur disque (2ème transfert).
le coût est 2 x T.

➢ Suppression **:** On cherche d'abord l'enregistrement, donc charger sa page (1 transfert), ensuite, on fait la modification en mémoire et on réécrit la page le coût est **2 x T**.
Si la suppression se fait par valeur (ex : supprimer tous ceux dont le nom est toto) :
- balayer tout le fichier (PxT) ;
- s'il y a n personnes répondant au nom toto n ≤ P ;
- si chacune se trouve dans une page distincte, il nous faut alors modifier n pages, puis les réécrire n x T donc le coût est : **(P x n) x T**.

On a supposé que les pages ne sont pas compactées. Et on donne plutôt que l'IdE, une condition sur le(s) enregistrement(s) à supprimer ?
Fichiers triés
➢ Balayage : Le coût est **P x T**.

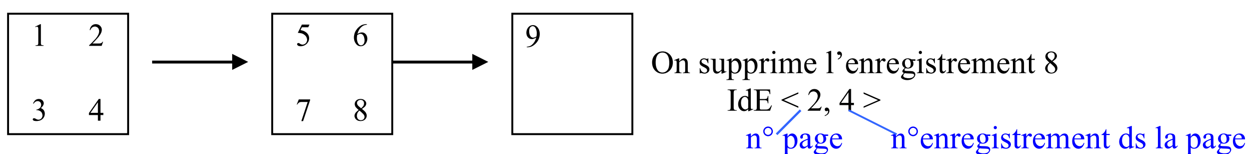
➢ Recherche avec égalité :
1. Même coût que le fichier tas, si le champs de recherche n'est pas celui du tri.
2. Sinon, avec une recherche dichotomique, on peut retrouver l'enregistrement en $\log_2(P)$ transferts le coût est **$\log_2(P)$ x T**.
Ici, on a considéré qu'un seul enregistrement trouvé.
Dans le pire des cas, par une recherche dichotomique :

1. on se place au milieu du fichier (lire page P/2). On restreint le fichier à P/2 pages.
2. on se place à 3P/4 donc $P/2^2$
n. il reste $P/2^n$ pages = 1 $\Rightarrow P/2^n = 1 \Rightarrow P = 2^n \Rightarrow \log P = \log 2^n \Rightarrow \log_2 P = n$

➢ Recherche avec intervalle **:** On fait d'abord une recherche dichotomique jusqu'à trouver une valeur dans l'intervalle spécifié ; le coût est **$\log_2(P)$ x T**. Là aussi, on a considéré qu'un seule valeur. S'il y a *x* enregistrements, alors il nous faudra transférer au maximum (X/R + 1) autres pages. Le coût est majorée par **.($\log_2(P)$ + X/R + i) x T.**
Chaque page contient E enregistrements donc X/E.
X/E + 1 page où X est le nombre d'enregistrements qui satisfont la condition. La recherche se fait en deux parties :
1. Trouver la valeur dans l'intervalle (ex : les valeurs entre 15 et 25 ans). Au pire des cas, on fait log2(P) x T. Au meilleur des cas, on ne fera qu'une seule lecture (la page du milieu contient une valeur comprise dans l'intervalle).
2. Ensuite, on recherche les x valeurs restantes. Celles-ci se trouvent sur au maximum X/E pages : on doit donc lire X/E pages.

➢ Insertion : Il faut d'abord chercher la bonne page où le placer. Ceci peut se retrouver en lisant $\log_2(P)$ pages. On peut supposer qu'en moyenne la page intéressante se trouve au milieu du fichier. Comme on risque d'avoir besoin d'avancer tous les enregistrements dans les pages suivantes P/2, on doit donc toutes les lire et les réécrire. Ce qui donne un coût global approximatif de **.($\log_2(P)$ +2 x P/2) x T.** Le coût est donc **.($\log_2(P)$+P) x T**.

➢ Suppression **:** On retrouve la page de l'enregistrement en un transfert. Ensuite on charge les pages suivantes pour tasser les enregistrements. Ce qui fait un coût global approximatif de **(1 + P/2) x 2 x T**. En fait cela dépend du nombre de page (pair ou impair). Le coût est **P x T**



Si l'on veut compacter, il faut placer le 9 à la place du 8 (coût supplémentaire). Donner une estimation du coût si on veut avoir une organisation compacté. On suppose que la suppression se fait dans la page du milieu.
1. On lit la page du milieu, (T) puis celle qui vient juste après. P/2 et P/2 + 1
2. A partir de la page du milieu jusqu'à la dernière page, chacune sera lue et écrite une fois d'où un coût de (P/2) x (Tx2) $\Rightarrow$ P x T
Nombre de page lues chaque page sera lue et écrite
Dans l'estimation de ce coût, on n'a pas tenu compte du temps de la réorganisation des pages au niveau de la zone tampon.
7
Fichiers hachés
On suppose qu'il n'y a pas de débordement

➢ Balayage : Pareil que pour le fichier tas le coût est donc **P x T**

➢ Recherche avec égalité :
1. Si *X* est la valeur recherché, alors on peut savoir la page en calculant *h(X)* et on fait un accès direct : le coût est **.T.**
2. S'il y a plusieurs enregistrements, comme on a supposé qu'il n'y a pas de débordement, alors tous les autres enregistrements sont dans la même page.

➢ Recherche avec intervalle **:** On est obligé de balayer tout le fichier le coût est **P x T**

➢ Insertion : Le coût est **2T** on accède directement à la page

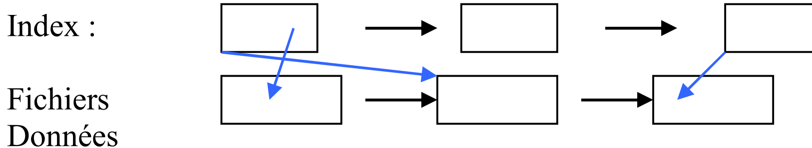
- **Suppression** : On lit la page puis on la réécrit le coût est donc **2T**

Conclusion
Il n'y a pas une organisation meilleure que toutes les autres pour tous les types d'opérations. Pour le balayage, c'est pareil, et pour la recherche, mieux vaut des fichiers hachés (ne tiens pas compte de débordement).

	Tas	Trié	Haché
Balayage	*P* x *T*	*P* x *T*	*P* x *T*
Recherche =	*0.5 (P* x *T)*	Log_2 *(P)* x *T*	*T*
Recherche []	*P* x *T*	Log_2 *(P)* x *T*	*P* x *T*
Insertion	*2* x *T*	*(*Log_2 *(P) + P)* x *T*	*2T*
Suppression	*[(P+1)* x *T] /2*	*P* x *T*	*2T*

II. LES INDEX

Un index est une structure annexe à un fichier. C'est très pratique pour la recherche mais il est difficile de faire les mises à jour lorsque l'on a beaucoup d'index.



- Il est caractérisé par une *clé de recherche K,*
- Il contient un ensemble *d'entrées* que l'on notera *K**.

Les entrées d'un index

Un enregistrement dans l'index constitue une entrée *K** permettant de retrouver efficacement le(s) enregistrement(s) ayant la valeur *K.* Trois structures possibles pour les *entrées :*

1. Enregistrement ayant la même structure que les enregistrements du fichier de données.
2. Une paire < K, IdE >
3. Une paire < K, liste_IdE >

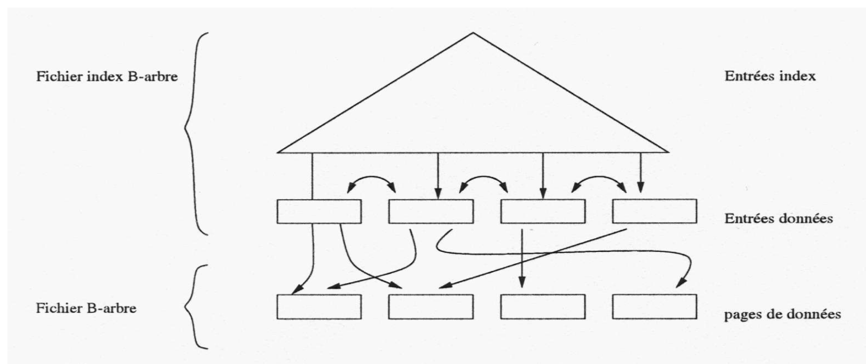
Si la structure de l'index est 1 alors pas besoin d'avoir en plus un fichier de données séparé.
Propriétés des index

Dense vs clairsemé

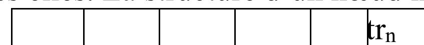
2.1) Index ISAM
Indexed Sequential Access Method

2.2) Arbres B+

Appelé B comme « Balancé » (équilibré en fait).Un arbre-B+ est caractérisé par sa **dimension** *d* qui est le nombre minimal de clés devant se trouver dans chaque nœud. *d* est la moitié du nombre maximal de clés qu'un nœud peut contenir. La structure des nœuds "feuille" est différente de celle des nœuds internes.
Les nœuds non terminaux sont des pointeurs vers des « feuilles » index et les feuilles sont des pointeurs vers



les données. Les feuilles sont liées entres elles. La structure d'un nœud interne est :



Si t est la taille d'un couple < Ptr, clé > et si T est la taille d'une page de l'index alors on prend généralement d=T/2t (chaque nœud doit être à moitié rempli) sauf pour la racine.

Coût d'une recherche dans un fichier indexé par un arbre B+

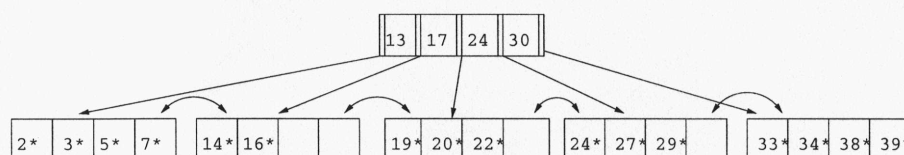
Comme l'arbre est équilibré, si *h* est la profondeur de l'arbre (le nombre de niveaux), le nombre de lectures qu'on doit faire avant d'atteindre une feuille c'est *h*. Ensuite, on accède directement à la page de données. D'où un coût total de clés dans le fichier de données. Au maximum, ces N clés sont stockés sur N/d feuilles. Avec chaque feuille contient 2xd clés au maximum et d clés au min. Le niveau *h* contient N/d feuilles au maximum et le niveau *h*- 1 contient (N/d)/d nœuds au maximum. On a N/d pointeurs au niveau *h* - 1 donc N/d paires <clé, Ptr>. Ces N/d paires sont réparties au maximum sur (N/d)/d nœuds. Le niveau h-2 contient N/d^3 nœuds au maximum.

Le niveau 1 contient un seul nœud au maximum.

$d^{h-1}=N \Rightarrow (h-1) \log d = \log N \Rightarrow h-1 = \log_d N \Rightarrow h= \log_d N+1$.

Le coût de la recherche avec égalité en utilisant un arbre B+ est de l'ordre de $\log_d N$ avec *d* qui est l'ordre de l'index et *N* le nombre de pages dans le fichier de données.

Exemple d'un arbre B+



Exemple d'un B-arbre d'ordre 2

Chaque nœud de l'index contient au moins 100 paires <clé, pointeur>. La recherche d'un enregistrement nécessite le parcours en profondeur de l'index.

$\log_d N$ lectures = log 1000 / log 100 = 3/2 ≈ 2

L'éclatement d'une feuille implique une remontée de la plus petite clé se trouvant dans la nouvelle feuille (dans l'exemple, c'est la clé 5). L'éclatement d'un nœud interne provoque le déplacement de la plus petite clef du nouveau nœud vers le nœud parents (dans l'exemple, la clé 17 a été déplacée vers un nœud parent).

Insertion de 8

Suppression de 19

Suppression de 20

nœud f n'étant pas à moitié plein:

1) 24 est ramené de fi vers f

L'entrée index 21 pointant vers fi remplace l'entrée 24 pointant vers fi

Dans cet exemple, la redistribution est possible car un des nœuds adjacents contient plus de *d* éléments.

Suppression de 24

voisin de f ne contient que 2 éléments, il n'est pas possible de faire de redistribution.

1) f est fusionné avec fi: 27* et 29* sont ramenées vers f
2) L'entrée index 27 pointant vers fi est supprimée du nœud m

Quand la fusion se fait avec le voisin à droite, ce sont les clés du voisin qui viennent dans **f** et l'entrée pointant vers fi est supprimée du nœud parent. Quand la fusion se fait avec le voisin gauche, ce Sont les clés de f qui Sont ramenées vers fi et l'entrée pointant vers f est supprimée du noeud parent.

Insertion

La dimension du B-arbre est *d*

1. Trouver la feuille f où *k* doit être insérée.
2. Si f n'est pas pleine alors y insérer *k*
   Sinon /* f est remplacée par f et f_1 */

mettre *d* entrées dans f et *d*+ 1 entrées dans f_1
insérer la *d* + 1ère clé dans le noeud *m* parent de f (appel récursif)
le fils gauche pointe vers f
le fils droit pointe vers f_1 Finsi

3. Au niveau du nœud index *m,* si *m* est plein alors remplacer *m* par *m* et m_1

Soit $k_1,\ldots,k_d,k_{d+1},k_{d+2},\ldots,k_{2d},k_{2d+1}$
mettre $\underbrace{k_1,\ldots,k_d}_{d}$ dans *m*
mettre $\underbrace{k_{d+2},\ldots,k_{2d+1}}_{d}$ dans m_1
insérer k_{d+1} dans le noeud parent de *m*

Suppression

1. Soit f la feuille où *k* se trouve
2. Supprimer *k* de f
3. Si f contient au moins *d* clés, alors FIN
4. Sinon

Si la redistribution avec les nœuds adjacents (le même parent) est possible, alors
faire la redistribution
Finsi
Sinon /* redistribution pas possible */
fusionner f avec un des nœuds adjacents
Supprimer du parent *m* de f un clé /*appel récursif */
Finsi

Création d'un arbre B+

Nous avons un fichier *F* pour lequel on veut créer un index sur la clé *K.* La première manière de procéder est de parcourir *F* et utiliser l'algorithme d'insertion dans un arbre B+ pour chaque clé lue. Le coût approximatif de ce procédé est le coût du parcours de *F* (i.e *N/d'* avec *d'* le nombre d'enregistrements qu'on peut mettre dans une page), plus le coût de l'insertion de chaque clé.
Soit *h* la hauteur de l'arbre B+, l'insertion d'une clé consiste à parcourir l'arbre à partir de la racine jusqu'à une feuille (i.e *h* lectures), ensuite l'écriture dans une feuille et sa recopie sur le disque (1 écriture). S'il y a éclatement, on est obligé de remonter d'un niveau et au pire, on doit remonter jusqu'a la racine, i.e 3h écritures. Le coût global est donc majoré par N/d'+(N*3*h).

Une autre manière de procéder consiste à :

1. d'abord trier les clés à insérer,
2. insérer un pointeur dans la racine vers la page la plus à gauche,

La suite de la procédure est expliquée sur l'exemple

Les pages du fichier triées sont les feuilles de l'arbre. Elles sont rentrées de gauche à droite, et l'insertion d une telle feuille conduit à l'insertion d'une clé dans le niveau supérieur de l'arbre. Cette dernière insertion peut être suivie d'un éclatement d'un nœud et l'augmentation d'un niveau de l'arbre.

Cette méthode est plus rapide que les insertions répétées.

2.3) Tri externe

C'est une opération très utile

- ORDER BY
- Le tri est utilisé lors de la création d'arbres B+
- Elimination de doublons (voir prochains cours)
- Des algorithmes de jointures l'utilisent (voir prochains cours)

Par exemple, pour trier un fichier de 10 Go quand on a une RAM de l00 Mo
Tri par fusion simple

Utilise 3 zones tampons

1. première étape: lire une page, la trier, et le réécrire (une seule zone tampon est utilisée)
2. Les étapes suivantes : les 3 zones tampons sont utilisées

- A chaque étape, nous lisons et écrivons chaque page du fichier.

- Si *N* est le nombre de pages, alors le nombre d'étapes est égal a $[\log_2 N] + 1$

- ainsi, le coût total est de

$$2N ([\log_2 N] + 1)$$

Tri par fusion général

Utiliser plus de 3 zones tampon (ce qui est généralement possible) pour trier un fichier a *N* pages quand on a *B* zones tampons.

1. étape 1 : utiliser les *B* zones ce qui donne en sortie *[N/B]* "sous fichiers" triés et chacun ayant *B* pages (sauf peut être le dernier qui en contiendra moins).
2. étapes suivantes, on fusionne *B* - 1 "sous fichiers".

Coût du tri par fusion général

Nombre d'étapes : $1+ [\log_{B-1} N]$. Le coût est donc 2N* nombre d'étapes.

Par exemple, avec 5 tampons et un fichier de 108 pages :

1. 1ère étape : [108/5] # 22 "sous fichiers" triés de 5 pages chacun (le dernier en contient 3).

2. 2ème étape : [22/4] # 6 "sous-fichiers" de 20 pages chacun (le dernier en contient 8).

3. 3ème étape : 2 sous fichiers, un de 80 pages et l'autre ayant 28.

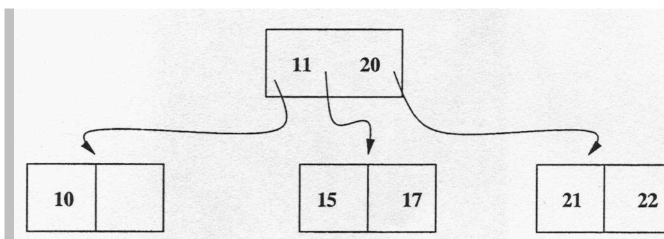
4. 4ème étape : 1 fichier trié de 108 pages.

Nombre d'étapes pour différentes valeurs de N et B :

	B=3	**B=5**	**B=9**	**=17**	**=129**	**=257**
	7	4	3	2	1	1
0	10	5	4	3	2	2
00	13	7	5	4	2	2
000	17	9	6	5	3	3
0000	20	10	7	5	3	3
00000	23	12	8	6	4	3
000000	26	14	9	7	4	4
0000000	30	15	10	8	5	4

Pour obtenir le coût, il suffit de multiplier le nombre d'étapes par 2N
EXERCICES

<u>3.1) Soit l'arbre B+ d'ordre 1 suivant :</u>



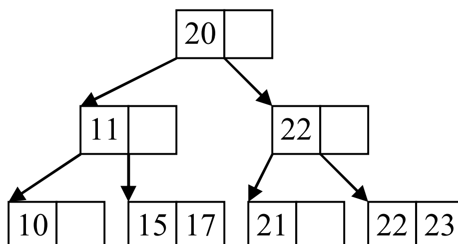
Donner les configurations consécutives de l'arbre après :

1. l'insertion de la clé 23 puis
2. la suppression de la clé 10.

1.
a) répartition des clés {21, 22, 23} (ici d=1). Les d premières, on les garde dans l'ancienne feuille f. Les d+1 suivantes dans la nouvelle feuille f_1.

d = (nombre max de clé par feuille) / 2

b) Remontée de 22 (plus petite clé se trouvant dans f_1).
c)



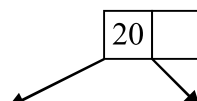
Pointeur gauche : < pointeur droit : ≥

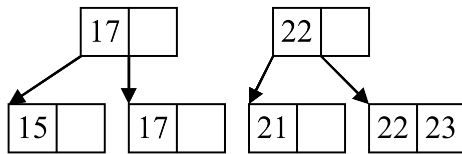
f f_1

Les clés présentes au niveau du fichier données sont toutes présentes au niveau des feuilles de l'index. Les clés se trouvant dans les nœuds internes de l'index ne sont pas nécessairement présentes dans le fichier des données. Dans l'exercice, nous avons 11 qui apparaît dans un nœud interne mais pas au niveau des feuilles : on peut en conclure qu'à un certain moment, 11 a été inséré puis supprimé.

2. f contient moins de d clés. On va essayer de faire une répartition avec les feuilles adjacentes qui est f1. La répartition est possible : on déplace 15 vers f, on remplace 11 par 17.

(on évite de supprimer des feuilles).





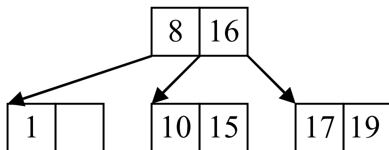
f f_1

3.2) Insertion d'une clé dans un autre arbre

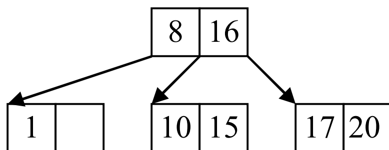
L'arbre B+ ci-dessous a été obtenu suite à l'insertion d'une clé dans un autre arbre. On nous dit aussi que la dite insertion a provoqué *l'éc1atement* d'un nœud de l'arbre.
Donner une configuration possible de l'arbre d'origine avec la clé qui y aurait été insérée. Cette configuration est-elle unique ? Si ce n'est pas le cas, alors donner toutes les configurations possibles de l'arbre d'origine.

La valeur qu'on a inséré doit se trouver au niveau des feuilles : {1, 10, 15, 17, 19 ou 20}. Comme il y a eu un éclatement, on en déduit une remontée d'une clef se trouvant au niveau des feuilles {19}. On en conclut aussi que c'est la feuille f qui a été éclatée en deux. Lors de l'insertion, on s'est retrouvé avec les clés {17, 19, 20}. Initialement dans f, on aurait pu avoir

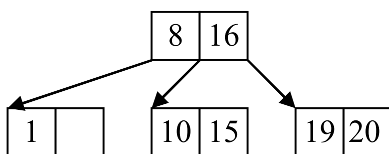
1. <u>17 et 19 et insertion 20</u>



2. <u>19 et 20 et insertion 17</u>



3. <u>17 et 20 et insertion 19</u>



<u>3.3) Coût de création d'un arbre B+</u>

Donner un estimation du coût de création d'un arbre B+ en fonction de N, de B et de M sachant que le coût comprend le coût de création et le coût du tri : $2M\,[1+ \log_{B-1} (M/B)]$.
N : nombre d'enregistrements,
B : nombre de zones tampons,
M : nombre de pages du fichier.
La hauteur de l'arbre principal est donnée par $\log_d N$ où d est la dimension. L'arbre final contient un nombre total de nœuds qui est approximativement égal à :
Nombre de nœuds feuilles = N/2d (toutes les feuilles sont pleines).
Nombre de nœuds au niveau supérieur (sachant que l'on a besoin de N/2d pointeurs et que chaque nœud contient au minimum d+1 pointeurs) = (N/2d) / (d+1).
Nombre de nœuds au niveau supérieur = $(N/2d) / (d+1)^2$
Le nombre total de nœuds :

$$(N/2d) \sum^{h} 1/(d+1)^i$$

i=1

Transactions

I. LE CONCEPT DE TRANSACTION

E

Une transaction est *unité de programme* qui accède aux données de la base en lecture et/ou écriture. Une transaction accède à un état cohérent de la base. Durant l'exécution d'une transaction, l'état de la base peut ne pas être cohérent. Quand une transaction est *validée* (commit), l'état de la base **doit** être cohérent.
Deux types de problèmes :

- problèmes systèmes (récupérabilité)
- exécutions concurrentes de plusieurs transactions (séralisabilité)

En fait , la seule condition est qu'à la fin de la transaction, la base soit cohérente. C'est la récupérabilité (crash system, erreurs = état incohérent : comment revenir à un état cohérent ?).

Propriétés
Pour préserver l'intégrité des données, le système doit garantir :

- **Atomicité** : Soit toutes les opérations de la transaction sont validées, ou bien aucune opération ne l'est.

- **Cohérence** : L'exécution d'une transaction préserve la cohérence de la base (c'est à l'utilisateur de vérifier).

- **Isolation** : Même si plusieurs transactions peuvent être exécutées en concurrence, aucune n'est censée prendre en compte les autres transactions i.e pour chaque paire de transactions *Ti*, *Tj*, pour *Ti* tout se passe comme si *Tj* s'est terminée avant le début de *Ti* ou bien qu'elle a commencé après la fin de *Ti* (les résultats intermédiaires de *Tj* ne lui sont pas apparents).

- **Durabilité :** Si une transaction est validée, alors tous les changements qu'elle a faits sont persistants (même s'il y a un crash).

*Ce sont les propriétés **ACID** (Atomicité, Cohérence,…)*

Exemple

Une transaction qui transfère 1000 Frs du compte *A* vers le compte *B*

Il faut essayer d'exécuter le maximum de transactions en même temps, tout en conservant la propriété d'isolation.

1. *Lire(A)*
2. *A:=A-1000*
3. *Ecrire(A)*
4. *Lire(B)*
5. *B:=B+1000*
6. *Ecrire(B)*

La base est cohérente si la somme $A + B$ ne change pas suite a l'exécution de la transaction (cohérence). Si la transaction "échoue" après l'étape 3, alors le système doit s'assurer que les modifications de *A* ne soient pas persistantes (atomicité). Une fois que l'utilisateur est informé que la transaction est validée, il n'a plus à s'inquiéter du sort de son transfert (durabilité).
Si entre les étapes 3 et 6, une autre transaction est autorisée a accéder a la base, alors elle "verra" un état incohérent *(*$A + B$ est inférieur à ce qu'elle doit être). L'isolation n'est pas assurée. La solution triviale consiste a exécuter les transactions en séquence.

II. LES ETATS D'UNE TRANSACTION

- **Active :** la transaction reste dans cet état durant son exécution.
- **Partiellement validée :** juste après l'exécution de la dernière opération.
- **Echec :** après avoir découvert qu'une exécution "normale" ne peut pas avoir lieu.

➢ **Avortée :** Après que toutes les modifications faites par la transaction soient annulées (Poli back). Deux options :

- Ré-exécuter la transaction
- Tuer la transaction

➢ **Validée :** après l'exécution avec succès de la dernière opération

Implémentation de l'atomicité

Approche naïve : c'est le *mécanisme de reprise sur panne.*

La notion de *copie* (shadow database) :

- On suppose qu'une seule transaction peut s'exécute en même temps.
- Un pointeur **pointeur_bd** pointe vers la version cohérente courante de la base.
- Toutes les mises a jour sont exécutées sur une copie. **Pointeur_db** ne pointera sur la copie que si la transaction est validée.
- Si la transaction échoue, alors la copie est supprimée. Si la transaction est validée, on garde la nouvelle sinon le pointeur repasse sur l'ancienne version.

Inefficace si la base est volumineuse !

III. EXECUTIONS CONCURRENTES

Plusieurs transactions peuvent être exécutées en concurrence pour :

➢ une meilleure utilisation du processeur (une transaction peut utiliser le processeur pendant qu'une autre accède au disque)

➢ la réduction du temps de réponse aux transactions (une transaction courte n'a pas à attendre la fin d'une longue transaction)

Le contrôle de la concurrence est un mécanisme permettant l'interaction entre transactions tout en assurant l'intégrité de la base.

3.1) Ordonnancement

En anglais : Schedule. En fait, c'est une séquence chronologique spécifiant l'ordre d'exécution d'opérations de plusieurs transactions.

Exemple

e(A)	
= A - 1000	
ire(A)	
e(B)	
= B + 1000	
ire(B)	
	e(A)
	*p := A * 0,1*
	ire(A)
	= A - Temp
	ire(A)
	e(B)
	= B + Temp
	ire(B)

C'est un ordonnancement "en série" de T_1 et T_2. On l'appellera O_1.

L' ordonnancement 03 représente une exécution "entrelacée" de T_1 et T_2. Il est équivalent *a* $<T_1,T_2>$.

¡ Sérialisabilité I

Dans la suite, on ne va considérer que les opéra-
L'ordonnancement suivant *(04)* ne préserve pas la valeur de $A + B$.

T_1
T_2
tions de lecture et d'écriture.

e Hypothèse Chaque transaction prise a part préserve la cohérence de la base.
LireÇA)
A := *A* - 1000
Ecrire(A)
Lire(B)
B := *B* + 1000
Ecrire(B)
Lire (A)
Temp A * 0, 1
A := A - Temp
Ecrire(A)
Lire(B)

e Ainsi, l'exécution en série préserve la cohérence.
- Un ordonnancement entrelacé est *sérialisable* s'il est équivalent *a* un ordonnancement en série.

Difrérentes définitions d'équivalence
B := B + Temp
Ecrire(B)
- c-sérialisabilité (Sérialisabilité de conflit)
- v-sérialisabilité (Sérialisabilité de vue)

I **c-sérialisabilité I**

- Les instructions f~ et des transactions *T~* et Tj sont en conflit s'il existe un objet Q accédé par t~ et et l'une d'elles écrit Q. Si t~ *Lire(Q)* et = *Lire(Q)* alors il n'y a pas de conflit.

- Si un ordonnancement *O* peut être transformé en *0'* par une série de remplacements *(swaps)* d'instructions non conflictuelles, alors *O* et *0'* sont *c-équivalents.*

I **c-sérialisabîlité I**

L'ordonnancement 03 ci-dessous peut être transformé en O~. Il est donc c-sérialisable.

O est c-sérialisable s'il est c-équivalent a un ordonnancement en série.

T_1	T_2
Lire(A)	
Ecrire(A)	
	Lire(A)
	Eerire(A)
lire(B)	
Ecrire(B)	
	lire(B)
	Bcrire(B)

b L'ordonnancement ci-dessous n'est pas c-séria lisable

T_3	$1T_4$
Lire(Q)	
Ecrire(Q)	
Bcrire(Q)	

j **v-sérialisabilité**

j **v-sérialisabilité** **I**

O et *O'* sont "v-équivalents en vue" Si

Pour chaque objet Q, Si dans *O,* ~ lit la valeur initiale de Q alors ~ lit la valeur initiale de Q dans *O'.*

Pour chaque Q, Si dans *O,* ~ lit une valeur de Q produite par ~, alors dans *O',* cette lecture doit aussi correspondre a une valeur produite par ~.

Pour chaque **Q, Si** dans *O,* ~ est la dernière a exécuter *Ecrire(Q),* alors elle est aussi la dernière a l'exécuter dans *O'.*

e *O* est *v-sérialisable* s'il est v-équivalent â un ordonnancement en série

e Chaque ordonnancement c-sérialisable est v-sérialisable

L'ordonnancement suivant est v-sérialisable

T_5 *jT_6* *jT7*

Lire(Q)
Ecrire(Q)
Ecrire(Q)
Ecrire(Q)

Il est v-équivalent a <*Ts,T_6,T_7*>

e Si *O* est v-sérialisable et non c-sérialisable, alors il contient des mises a jour sans ef~t

j Autres notions de sérialisabilité i

- L'ordonnancement ci-dessous est équivalent a <*Ti,T*> pourtant il n'est ni v-sérialisable ni c-sérialisable

T_1
Lire(A>
A := A - **1000**
Ecrire(A)

Lire(B)
B := B + **1000**
Ecrire(B)
T

Lire(B)
B := B - 10
Ecrire(B)

Lire(A)
A *A* + 10
Ecrire(A)

Reprise sur panne. Récupérabilitél

e Ordonnancement *récupérable* Si *T~* lit un objet précédemment écrit par Tj alors la validation de T, a lieu avant la validation de T'~

- L'ordonnancement suivant n'est pas récupérable **Si** *T_9* valide tout de suite après la lecture

T_8 *jT_9*

Lire(A)
Eerire(A)
Lire(A)
Ecrire(B)
Si T_8 doit être avortée, alors T_9 aura lu une valeur qui peut être "incohérente"

- Pour déterminer ce type d'équivalence, il faut analyser des opérations autres que *Lire* et *Ecrire.*

Le SGBD doit s'assurer que les ordonnancements soient récupérables.

Récupérabilité (suite)

L'echec d'une transaction peut conduire a l'avortement de plusieurs transactions

T,.
T_{10} T_{11} T_{12}
Lire(A)
Lire(B)
Ecrire(A)
Lire(A)
Ecrire(A)
Lire(A)
Si T_{10} échoue, alors T_{11} et T_{12} doivent être avortées.

Récupéra bilité (suite) I

e Ordonnancement sans cascade : Si pour chaque paire ~ T, t q T, lit une donnée précédemment écrite par ~, alors la validation de T~ a lieu avant celle de

e Un ordonnancement sans cascade de roîl back est récupérable.

e Il est souhaitable de restreindre les ordonnancements a ce ceux qui sont sans cascade

Implémentation de l'isolationI

- Les ordonnancements devraient être
v- ou c-sérialisables, récupérables pour garder la cohérence de la base et de préférence sans cascade.

- Si l'on autorise que les ordonnancements en série, alors toutes les propriétés sont garanties.

- Faire la balance entre le taux de concurrence et le traitement en plus qui s'y greffe.

Définition des transactions en SQLI

Dans SQL, une transaction commence implicitement.

Une transaction se termine soit par l'exécution d'unE commande **COMMIT** (ou COMMIT WORK) soit par ROLL BACK (ou ROLL BACK WORK)

Exemple de testi

T_2 T_3 T_1 T_4 T_5
Tester la sérialisabilitél
Li~e(X)

Considérer un ordonnancement *O* des transactions T_1,. . .*T'~*. Le graphe de précédence de *O* est un graphe *(N, A)* où

Lire(Y)
Li~c(Z)

Lir~(U)
Lsre(Y)
Liee(V)
Lire(W)
ECTire(W)
Ecrire(Z)

- *N* est l'ensemble des transactions

- Il y a un arc (~<T,) s'il y a un conflit entre ~ et T, sur un objet Q et ~ accède a Q avant T,

O est c-sérialisable 55 son graphe de précédence est acyclique.

Lire(U)
Ecrsre(U)

Le r~(Y) Fcrer£(Y)
Lira(Z)
Ecrire(Z)

Test de la v-sérialisabilitéî

Il a été montré que le test de la v-sérialisabilité est un problème NP-complet. Il est donc très peu probable que l'on trouve un algorithme efficace (polynômial) qui puisse faire ce test.

Exemple de test

T_1 *j* T_2 *Lire(A)*
Lire(A)
Ecrire(A)
Ecrire(A)

Dans la pratique, on se contente de la c-sérialisabilit<

conclusion

- Tester Si un ordonnancement est sérialisable *après* son exécution (ou bien sur son graphe de précédence) est inefficace.

Contrôle de concurrencel

e Protocoles basés sur les verrous

e Protocoles basés sur les estampilles

- But Développer des stratégies de contrôle de concurrence qui puissent garantir la c-sérialisabilité. Ces protocoles n'auront pas besoin de faire le test sur le graphe de précédence (utilisation de techniques de verrouillage, cf prochain cours)

Pourquoi ce cours ? Pouvoir décider Si un protocole est correct par rapport â une notion de sérialisabilité.

e Protocoles basés sur la validation

e Différentes granularités

e Gestion des deadlock (verrous mortels)

Notion de verrouillage~

Les transactions *posent des verrous* sur les données auxquelles elles veulent accéder.

Les données peuvent être verrouillées de deux manières

Notion de verrouillage (suite)

e Matrice de compatibilité de verrous

- **Verrou exclusif Dans ce cas, la donnée** peut être lue et écrite. Le verrou *VX* est attribué lors de l'exécution de l'opération Lock~X(donnée)

Verrou partagé : Dans ce cas, la donnée ne peut être que lue. Le verrou *VP* est attribué suite a l'exécution de *Lock~P(donnée)*

Les demandes de verrous sont adressées au gestionnaire de la concurrence.

Une transaction ne peut avancer tant que le verrou qu'elle demande ne lui est pas attribué.
jipI *x*

½frfrP oui non
fififix non non

Une transaction ne peut poser un verrou sur une donnée que Si ce verroù est compatible avec les verrous qui y sont dèja posés.

- Lors de son exécution, une transaction peut libérer certains verrous. Utilisation de la commande *Unlock((*

Notion de verrouillage (suite)
e Considérer l'ordonnancement suivant

Notion de verrouillage (suite)

T1
T2

T1 : Lock~P(A)
Lire(A)
Unlock(A)
Lock~P(B)
Lire(B)

Unlock(B)
Afficlîer(A + B)

Un protocole de verrouillage est une discipline qui dicte aux transactions comment elles demandent et comment elles libèrent les verrous
Lock~X(B)
Lire(B)
B *:= B* - 1000 *Ecrire(B)*

Lock~X(A)
Ecrire(A)
LOck~P(A)
Lire(A)
Lock~P(B)

Lire(B)

T_1 et T_2 seront bloqués. Nous sommes en situation de *deadlock.* Pour résoudre ce problème, une des deux transactions doit étre avortée et ses verrous libérés

e La possibilité de deadlock existe dans presque tous les protocoles

Verrouillage a deux phasesi

verrouillage 2PL

b Ce protocole garantit la c-sérialisabilité Phase 1:

Verrouillage a deux phasesI

- La transaction peut poser des verrous

- Elle ne peut pas en libérer

- Phase 2:**e** Le 2PL ne garantit pas l'absence de deadlocks **e** Le 2PL ne garantit pas l'absence de

cascades

- La transaction peut libérer des verrous
- Elle ne peut plus en poser

On associe a chaque transaction un point de verrouillage qui correspond au moment où elle pose son dernier verrou (la fin de la première phase). On montre alors que les ordonnancements sont équivalents a l'exécution en série selon l'ordre des points de verrouillage des transactions.

Extension : "2PL stricte" et "2PL rigoureux"

Acquisition des verrousi

Extension du 2PLI

- Dans le 2PL stricte, les transactions gardent leurs verrous exclusifs jusqu'au commit.

- Dans le 2PL rigoureux, les transactions gardent **tous** leurs verrous jusqu'au commit.

La différence *: T~,* qui vient après T,, peut écrire un objet *A* après que T, l'ait lu et avant que T, ne soit validée.
Alors que dans le 2ème cas, ~ ne peut pas modifier un objet accédé par tant que celle-ci n'a pas validé.
C'est généralement par â l'utilisateur d'inclure dans son code les demandes de verrous.
Si une transaction *T~* veut lire/écrire un objet *D* sans demander explicitement un verrou, alors l'algo suivant est utilisé

Pour l'opération de lecture

Si ~ a un verrou sur *D* **Alors**
Lire(D)
Sinon
Tant que il y a un transaction avec un verrou
X **sur** *D* **Faire**
Attendre
Fournir un verrous S a ~ sur *D* Lire(D)

Protocole avec estampillage~

j **Acquisition des verrousi**

Pour l'opération d'écriture

Si ~ a un verrou X sur *D* **Alors** Ecrire(D)
Sinon
Tant que il y a une transaction avec
un verrou sur *D* **Faire**
Si T'~ a déja un verrou S sur *D* **Alors**
Transformer S en X
Sinon

Fournir un verrou X sur *D* a ~ Ecrire(D)
Le but est d'avoir des ordonnancements sérialisables équivalents à l'ordre chronologique des transactions

e **A** chaque transaction est associée une estampille relative au moment où elle "arrive", i.e T'~ avant T, *~ ST'~) < ST(T,)* (l'heure système ou bien un simpIe compteur)

e **A** chaque objet *D* de la base sont associées 2 estampilles
On ne dit pas quand est-ce que les verrous sont libérés !!
1. *E~ST'(D)* **:** la plus grande des estampilles des transactions qui ont écrit *D* avec succès
2. *L~sT(D) :* la plus grande des estampilles des transactions qui ont lu *D* avec succès

Protocole avec estampillage~
j **Protocole avec estampillagel**
Supposons que ~ veuille lire *D*
Si *ST(T) > E~sT(D),* alors la lecture est autorisée et
L~ST(D) l max{L~ST'(D),ST~)}
Supposons que ~ veuille écrire *D*

e Si *TS~) < L~sT(D) :* dans ce cas, cela veut dire qu'il y a une transaction qui est arrivée après T et qui a lu *D.* *T* est annulée

e Si *TS(T) < E~sT'(D)* si on laisse faire cette écriture, alors elle va "écraser" celle faite par une transaction arrivée après T *T* est annulée
- Si *ST(T) < E~ST(D),* alors *T* est "annulée" et relancée avec une nouvelle estampille

e Sinon,
- l'écriture est réalisée
- *E~sT(D) ' ST(T)*

j Protocole avec estampillage~
Soient les transactions T_1, T_2, T_3, T_4 et T_5 avec les estampilles resp. 1, 2, 3, 4 et 5.
L'ordonnancement suivant représente une situation où T_2 et T_4 sont annulées

T_1 T_2 T_3 T_4 T_5
Lire(X)
Lire(Y)
Lire(Y)
Lire(Y)
Ecrire(Y)
Lire(Z)
Ecrire(Z)
Lire(Z)
Lire(Z)
abort
Lire(X)
Ecrire(Z)
abort
Ecrire(X)

*y*____
____________________ *Ecrire(Z)*

j **Protocole avec estampillage~**

e L'estampillage permet d'éviter les blocages (les transactions sont exécutées ou bien annulées)

e Il garantit la c-sérialisabilité puisque tous les arcs sont de la forme *T "* T, avec *ST(T) <ST(T,)*

e Par contre, le problème de récupérabilité persiste. Si *T* est annulée alors que T, a lu une valeur écrite par *T~* alors T, doit aussi être annulée. Si T, a déja validé, alors l'ordonnancement n'est pas récupérable.

¡Protocole basé sur la validationî

L'exécution d'une transaction *T* se fait en 3 phases

Lecture : Les écritures se font dur des "variables
locales"

Protocole basé sur la validationi

Pour faire le test de validité, nous avons besoin de savoir a quels moments ont lieu les différents phases. D'où l'utilisation d'estampilles

1. Début(T~): Le moment où *T* a débuté

Validation : Tester la validité des écritures (ne violent elles pas la sérialisabilité ?)

Ecriture : Si la validation réussit, alors les écritures sont retranscrites sur la base

Protocole basé sur la validationi

2. **Validation(T~): Le** moment où *T* a terminé sa phase précédente

3. **Fin(T~): Le** moment où *T* termine la phase d'écriture

La sérialisabilité est testée en se basant sur un estampillage des transactions correspondant a leur estampille de validation, i.e *ST(T) = Validation(T).*

Exemple d'ordonnancementl

La validation de T, réussit Si pour chaque *T* t.q
ST(T) <ST(T,)

T_1
Lire(B)

b *Fin(T) < Débat(T,),* ou bien

• *Débat(T~) < Fin(T~ < validation(T,)* et l'ensemble des données écrites par *T* est distinct de celui des données lues par T,

Pour le 2ème point:

• Les écritures de *T* n'ont pas d'effet sur les lectures de *T~* (elles ont lieu après la fin des lectures de *T)*

• les écritures de *T* n'ont pas d'effet sur les lectures de T, (T, ne lit aucun objet écrit par *T)*

Lire(A)
<valider>
Affieher(A + B)
T_2

Lire(E)
B := B - 50
Lire(A)
A := A + 50

<valider>
Ecrire(E)
Ecrire(A)

Note: La méthode de validation permet d'éviter les cascades de Poli back puisque les lectures ne se font que sur des données persistantes.

Granularité du verrouiIlage~

Granulanté du verrouiIlage~

Nous considérons la hiérarchie suivante

• Nous sommes en présence d'une structure d'arbre de

la forme

Base
Table **Table4** **Table n**

Page
{
Tuple

Permettre aux transactions de verrouiller n'importe quel niveau mais en respectant un nouveau protocole
Avant de verrouiller un objet *D, T* doit avoir un verrou intentionnel sur tous les ancêtres de *D*

___'[IPIXI P]IX[PIXI
IP ~ oui oui oui ~f oui
IX ~ oui f oui ___ ~f ___
P oui __ oui ~
X ~_~_ _ __ PIX oui __ __ 113
49

Page Il Page tm
Tuptelti TupleII2

e Un noeud *D* peut être verrouillé en *P* ou *IP* Si le parent est déja verrouillé en *IX* ou
• *D* peut être verrouillé en *X, PIX* ou~ Si le parent de *d* est verrouillé en *IX* ou *PIX*
T peut libérer un verrou sur *D* si elle ne possède plus de verrous sur les descendants de *D*
La aussi, on utilise le verrouillage en deux phases

Gestion des bIocages~
Granularité du verrouiIIage~

Exemples
Prévenir vs guérir
1) Prévenir

T_1 parcourt *R* et met a jour quelques tuples
• T_1 obtient un verrou *PIX* sur *R, a* chaque lecture d'un tuple, elle pose d'abord un verrou *P,* et Si elle a besoin de le modifier, elle transforme le *P* en *X*
T_2 utilise un index pour lire une partie de *R*
• T_2 pose un verrou *IP* sur *R,* ensuite elle obtient des verrous *P* pour chaque tuple qu'elle veut lire
T_3 lit la table *R* en entier
Soit *T* qui demande un verrou en conflit avec celui déja détenu par T,

e Privilégier les plus anciennes transactions (estampillage):

- *ST(T) < ST(T,) ~ T* peut attendre T,
- *ST(T) > ST(T,) ~ T* est annulée

- Si *T* est relancée avec une nouvelle estampille, alors elle risque d'attendre longtemps!! La relancer avec la même estampille.

- Noter qu'ici, seules les transactions demandeuses sont annulés
e Privilégier la transaction qui détient le verrou
• T_3 demande un verrou sur *R* ou bien
- *ST(T) <ST(T,) ~ T.* est annulée
-*ST(T) > ST(T,)* alors *T* attend

Gestion des blocages
Prévenir vs guérir

1) <u>Guérir</u>

<u>Gestion des blocages</u>

Ici, il faut détecter le blocage. Le système maintient un graphe G *(N, A)* avec

Prévenir vs guérir

Que faire Si un blocage est détecté ?

N les transactions

- *T ' T,* 55 *T* attend que T, libère un verrou

- Quand *T* demande un verrou détenu par T,, alors l'arc *T '* T, est r ajouté au graphe

Il faut annuler une transaction participant au cycle.

Choisir celle qui permet de réduire au maximum le nombre de cycles

- *T '* T, est supprimé quand T, ne détient plus le verrou demandé par *T*

Choisir celle qui est la moins proche de son état de validation

- Le système est bloqué ssi G contient un cycle
- Un algorithme est lancé périodiquement pour tester 'acyclicité

<u>Le problème</u>

La suppression d'ùn tuple ne peut se faire que Si *T* détient un verrou *X* sur ce tuple
L'insertion d'un tuple par T, implique la détention d'un verrou *X* par T, sur ce tuple
T veut supprimer tous les comptes dont le solde est
\> 300. Elle verrouille donc tous ces tuples.
T, insère un compte avec un solde > 400. Noter qu'il n'y a pas conflit.

- Ensuite, *T* affiche les comptes dont le solde est > 200. Le nouveau tuple est affiché.

Noter que cette exécution n'est pas équivalente a une exécution en série pourtant l'ordonnancement est c-sérialisable!!
Choisir une transaction qui n'a pas été annulée plusieurs fois